INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A verification algorithm for Declarative Concurrent Programming

Jean Krivine

N° ????

June 2006

Thème ?

ISSN 0249-6399

A verification algorithm for Declarative Concurrent Programming

Jean Krivine

Thème ? —
Projet MOSCOVA

Rapport de recherche n° ???? — June 2006 — 15 pages

Abstract: A verification method for distributed systems based on decoupling forward and backward behaviour is proposed. This method uses an event structure based algorithm that, given a CCS process, constructs its causal compression relative to a choice of observable actions. Verifying the original process equipped with distributed backtracking on non-observable actions, is equivalent to verifying its relative compression which in general is much smaller. We call this method Declarative Concurrent Programming (DCP).

DCP technique compares well with direct bisimulation based methods. Benchmarks for the classic dining philosophers problem show that causal compression is rather efficient both time- and space-wise. State of the art verification tools can successfully handle more than 15 agents, whereas they can handle no more than 5 following the traditional direct method; an altogether spectacular improvement, since in this example the specification size is exponential in the number of agents.

Key-words: Process algebra, transaction, event structures, verification, bisimulation

Un outil de vérification pour la Programmation Concurrente Déclarative

Résumé : Nous proposons une méthode de vérification pour les systèmes distribués basé sur la distinction entre comportement avant et arrière d'un système transactionnel. Cette méthode utilise un algorithme basé sur les structures d'événements qui, étant donné un processus CCS, construit son système de transition causal relatif à un ensemble d'actions observables. La vérification du processus CCS d'origine, équipé d'un mécanisme de retour arrière sur les transitions non observables, revient à vérifier la correction du système de transitions causales du processus qui est en général beaucoup plus petit. Cette méthode est appelée programmation concurrente déclarative (PCD).

Les performances de la PCD comparées aux performances des techniques traditionnelles de bisimulation donnent des résultats encourageants. Un banc d'essai utilisant le problème classique du dîner des philosophes montre que la PCD est plus efficace que la méthode directe, à la fois en terme de temps et d'espace de calcul requis. En effet, les outils standard de bisimulation peuvent vérifier des systèmes allant au delà de 15 philosophes dans le cas de la PCD, alors qu'ils ne peuvent gérer plus de 5 philosophes avec un approche de programmation directe. Cet amélioration des performances est d'autant plus spectaculaire que la taille du système de spécification des philosophes est exponentielle dans le nombre d'agents.

Mots-clés : Algèbres de processus, transactions, structures d'événements, vérification, bisimulation

1 Introduction

Backtracking is commonplace in transactional systems where different components, such as processes accessing a distributed database, need to acquire a resource simultaneously. To ensure unconditional correctness of the overall execution of the transaction, one usually provides a code that incorporates explicit escapes from those cases where a global consensus cannot be met. Such an upfront method generates a large and unstructured state space, which often means verification based on proving that the code is bisimilar to a reference specification becomes unfeasible.

Based on earlier work, we propose here an indirect verification method, and show on an example that it can handle larger specifications. The idea is to break down the distributed implementation of a given reference specification in two steps. First, one writes down a code which is only required to meet a weaker condition of causal or forward correctness relative to the specification. This condition is parameterised by a choice of observable actions corresponding to the actions of the specification. Second, the obtained code is equipped with a generic form of distributed backtracking on non-observable actions. A general theorem reduces the correctness of the latter partially reversible code to the causal correctness of the former [1].

In many transactional examples, this structured programming method works well, and obtains codes which are smaller, and simpler to understand [2]. It also seems interesting from a correctness perpective, since one never has to deal with the full state space, and it is enough to consider the much smaller state space of the forward code causal compression relative to observable actions. Thus it obtains codes which are also easier to prove correct. It is only natural then to ask whether and to which extent such indirect correctness proofs can be automated. This is the question we address in this paper.

Specifically we propose an algorithm, which, under certain rather mild assumptions about the system of interest, will compute its causal compression relative to a choice of observables. The true concurrency semantics tradition of using event structures as an intrinsic process representation comes to the rescue here. Indeed, event structures provide a representation of computation traces up to trace equivalence, and therefore reduce redundancy during the search of the compression. Besides event structures are uniquely suited to the handling of causal relationships between various events triggered by a process [3]. For these reasons our procedure includes a translation of the process as a recursive flow event structure, and computes the relative causal compression on this intermediate representation. The algorithm also relies on a compact representation of the conflict relation between events, and seems to perform well both space-wise, obtaining a much smaller state space, and time-wise. Benchmarks given for the classical example of the dining philosophers show a significant state compression, and a relatively low cost incurred by compression. Direct programming generates a state space that is already too big for being constructed by bisimulation verifiers for 6 agents, whereas our method can go well beyond 15.

The language we use to formalize concurrent systems is the Calculus of Communicating Systems (CCS) [4]. This is a slightly more expressive language than basic models of communicating automata, in that processes can dynamically fork. On the other hand, this

Processes	p, q	$::=$	$a.p$	Action prefixing
			$p \mid q$	Parallel composition
			$p + q$	Choice
			$D(\tilde{x}) := p$	Recursive definition
			$(x)p$	Name restriction
			0	Empty process
Actions	a	$::=$	$x, y, \ldots$	Input
			$\bar{x}, \bar{y}, \ldots$	Output
			τ	Silent action

Figure 1: CCS syntax

communication model includes no name-passing, which is a severe limitation in some applications. As is discussed further in the conclusion it is possible to adapt the present development, which is largely independent of the chosen communication model, to richer languages such as π-calculus.

Section 2 starts with a quick recall of CCS [4]. Section 3 develops its reversible variant RCCS, together with the central notion of causal correctness, and the fundamental result connecting causal correctness of a CCS process and full correctness of its lifting as a partially reversible process in RCCS [1]. The relative causal compression algorithm, and the accompanying verification method are explained in Section 4. Section 5 compares this method with the traditional direct method, using the dining philosphers problem as a benchmark. The conclusion discusses related work and further directions.

2 CCS

2.1 Syntax

CCS processes interact through binary communications on named channels: an output on channel x is written $\bar{x}$, an input on the same channel is simply written x. The complete syntax is given in Fig 1.

We write P for the set of processes, A for the set of actions, and A^* for the free monoid of action words. Restriction $(x)p$ binds x in p and the set of free names of p is defined accordingly. In a recursive definition $D(\tilde{x}) := p$ free names of p have to be $\tilde{x}$.

2.2 Operational semantics

A *labelled transition system* (LTS) is a tuple $\langle S, s, L, \rightarrow \rangle$ where S is called the state space, s the initial state, L the set of labels, and $\rightarrow \subseteq S \times L \times S$ the transition relation. One uses

$$\frac{}{a.p + q \rightarrow_a p}(\mathsf{act})$$

$$\frac{p \rightarrow_a p' \qquad q \rightarrow_{\bar{a}} q'}{p \mid q \rightarrow_\tau p' \mid q'}(\mathsf{synch}) \qquad \frac{p \rightarrow_a p'}{p \mid q \rightarrow_a p' \mid q}(\mathsf{par})$$

$$\frac{p \rightarrow_a p' \quad a \notin \{x, \bar{x}\}}{(x)p \rightarrow_a (x)p'}(\mathsf{res}) \qquad \frac{p \equiv p' \rightarrow_a q' \equiv q}{p \rightarrow_a q}(\mathsf{equiv})$$

Figure 2: CCS labelled transition system

the common notation $s \rightarrow_a t$, and for $m = a_1 \dots a_n \in A^*$, $s \rightarrow_m^* t$ means $s \rightarrow_{a_1} s_1, \dots, s_{n-1} \rightarrow_{a_n} t$ for some states $s_1, \dots, s_{n-1}$.

The operational semantics of a CCS term p is given by means of such an LTS $(P, p, A, \rightarrow)$, written $\mathsf{TS}(p)$, where $\rightarrow$ is given inductively by the rules in Fig 2. The equivalence relation $\equiv$ is the classical structural congruence for choice and parallel composition, together with the recursion unfolding rule $\big(D(\tilde{x}) := p\big) \equiv p$.

2.3 Process equivalence

Several variants of observational equivalence for CCS processes have been considered. We use here a variant of weak bisimulation based on the choice of a countable distinguished subset K of the set of actions A, which we fix here once and for all. Actions in K are called *observable* actions. The complement $A \setminus K$ of non-observable actions is denoted by K^c and also taken to be countable.

Let $\mathcal{S}_1 = (S_1, s_1, A, \rightarrow)$ and $\mathcal{S}_2 = (S_2, s_2, A, \rightarrow)$ be LTSs both with labels in A, a relation $\mathcal{R}$ over $S_1 \times S_2$ is said to be a *weak simulation* between $\mathcal{S}_1$, $\mathcal{S}_2$, if $s_1 \mathcal{R} s_2$ and whenever $p_1 \mathcal{R} p_2$:
— if $p_1 \rightarrow_a q_1$, $a \in K^c$, then $p_2 \rightarrow_m^* q_2$ with $m \in (K^c)^*$, and $q_1 \mathcal{R} q_2$;
— if $p_1 \rightarrow_a q_1$, $a \in K$, then $p_2 \rightarrow_m^* q_2$ with $m \in (K^c)^* a (K^c)^*$, and $q_1 \mathcal{R} q_2$.

The idea is that $\mathcal{S}_2$ has to simulate the behaviour of $\mathcal{S}_1$ regarding observable actions, but is free to use any sequence of non observable ones in so doing. Such a relation $\mathcal{R}$ is said to be a *weak bisimulation* if both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are weak simulations. When there is such a relation, $\mathcal{S}_1$ and $\mathcal{S}_2$ are said to be *bisimilar*, and one writes $\mathcal{S}_1 \sim \mathcal{S}_2$.

A CCS process p is said to be a *correct implementation* of a specification LTS $\mathcal{S}$, if $\mathsf{TS}(p) \sim \mathcal{S}$. When the specification is clear from the context, we may simply say p is correct. One thing to keep in mind is that all these definitions are relative to a choice of K. Usually, K is taken to be $A \setminus \{\tau\}$, but this more flexible definition will prove convenient.

3 Reversible CCS

We turn now to a quick intuitive introduction to RCCS. Consider the following CCS process:

$$(x)\big(x \mid x \mid \bar{x}.\bar{x}.a.p \mid \bar{x}.\bar{x}.b.q\big) \tag{1}$$

Both subprocesses $a.p$ and $b.q$ require two communications on x to execute, so the whole process may reach a deadlocked state $(x)\big(\bar{x}.a.p \mid \bar{x}.b.q\big)$ where neither a nor b may be triggered. If the intention is that the system implements the mutual exclusion process $a.p + b.q$, a possible fix is to give both subprocesses the possibility to release x:

$$(x)\big(x \mid x \mid R_p(x,a) \mid R_q(x,a)\big) \tag{2}$$

with $R_p(x,a) := \bar{x}.\big(\tau.(R_p(x,a) \mid x) + \bar{x}.(\tau.(R_p(x,a) \mid x \mid x) + a.p)\big)$.

This example helps in realising two key things: first the original code (1) although not correct, is partially correct in the sense that any successful action a or b leads to a correct state p or q; second the proposed fix can be made an instance of a generic distributed backtracking mechanism. The idea of RCCS is to provide such a mechanism, in a way that partial or causal correctness (yet to be defined formally) in CCS, can be proved to be equivalent to full correctness of the same process once lifted to RCCS [5].

3.1 Syntax

RCCS *forward actions* are the same actions as CCS, namely A. Recall these are split into K and its complement K^c. In the RCCS context actions in K are also called irreversible, or sometimes commit actions (following the transaction terminology); actions in K^c are also called reversible, since these are the ones one wants to backtrack. RCCS therefore also has *backward actions* written a^-, with $a \in K^c$.

RCCS processes are composed of *threads* of the form $m \triangleright p$, where m is a *memory*, and p is a plain CCS process:

$$r ::= m \triangleright p \mid (r \mid r) \mid (x)r$$

Memories are stacks used to record past interactions:

$$m ::= \langle \theta, a, p \rangle \cdot m \mid \langle\!|\theta|\!\rangle \cdot m \mid \langle\rangle$$

where θ is a thread identifier drawn from a countable set. Open memory elements $\langle \theta, a, p \rangle$ are used for reversible actions and contain a thread identifier θ, the action last taken, and the alternative process that was left over by a choice if any. Closed memory elements $\langle\!|\theta|\!\rangle$ are used for irreversible actions, and only contain an identifier. The prefix relation on memories is defined as $m \sqsubseteq m'$ if there is an m'' such that $m'' \cdot m = m'$.

Processes are considered up to the usual congruence for parallel composition together with the following specific rules:

$$\begin{array}{lcll} m \triangleright \big(D(\tilde{x}) := p\big) & \equiv & m \triangleright p & \\ m \triangleright (p \mid q) & \equiv & (m \triangleright p) \mid (m \triangleright q) & \\ m \triangleright (x)p & \equiv & (x)(m \triangleright p) & \text{if } x \notin m \end{array}$$

$$\frac{a \in K^c \quad \theta \notin m}{m \rhd a.p + q \rightarrow_a^{\theta} \langle \theta, a, q \rangle \cdot m \rhd p}(\mathsf{act}) \qquad \frac{a \in K^c}{\langle \theta, a, q \rangle \cdot m \rhd p \rightarrow_a^{\theta^-} m \rhd a.p + q}(\mathsf{act}^*)$$

$$\frac{a \in K \quad \theta \notin m}{m \rhd k.p + q \rightarrow_k^{\theta} \langle\!|\theta|\!\rangle \cdot m \rhd p}(\mathsf{commit})$$

$$\frac{r \rightarrow_a^{\Theta} r' \quad \theta \notin s}{r \mid s \rightarrow_a^{\Theta} r' \mid s}(\mathsf{par}) \qquad \frac{r \rightarrow_a^{\Theta} r' \quad s \rightarrow_{\bar{a}}^{\Theta} s'}{r \mid s \rightarrow_{\tau}^{\Theta} r' \mid s'}(\mathsf{synch})$$

$$\frac{r \rightarrow_a^{\Theta} r' \quad a \neq x, \bar{x}}{(x)r \rightarrow_a^{\Theta} r'}(\mathsf{res}) \qquad \frac{r \equiv r' \rightarrow_a^{\Theta} s' \equiv s}{r \rightarrow_a^{\Theta} s}(\mathsf{equiv})$$

Figure 3: RCCS labelled transition system

Any CCS process p can be lifted to RCCS with an empty memory $\ell(p) := \langle\rangle \rhd p$, and conversely, there is a natural forgetful map φ erasing memories and mapping back RCCS to CCS. Clearly $\varphi(\ell(p)) = p$. When we want to insist that the lift operation is parameterised by the set K, we write $\ell_K(p)$.

3.2 Operational semantics

The operational semantics of RCCS is also given as an LTS with transitions given inductively by the rules in Fig 3. In the contextual rules Θ stands either for θ or θ^-. The freshness of the thread identifier θ is guaranteed by the side conditions $\theta \notin m$ in the (act) and (commit) rules, and $\theta \notin s$ in the (par) rule. The use of such identifiers makes the presentation given here somewhat simpler than the earlier one [1]. Note that backtracking as defined in the operational semantics is a binary communication mechanism of exactly the same nature as usual forward communication. However, since threads are required to backtrack with the exact same thread with which they communicated earlier, backtrack can be shown to be confluent, at least for those processes that are reachable from the lifting of a CCS process.

The (commit) rule uses a closed memory element $\langle\!|\theta|\!\rangle \cdot m$ indicating that the information contained in m is no longer needed, since by definition actions in K are not backtrackable. Supposing r is a process where any recursive process definition is guarded by a commit, an assumption to which we will return later on, this bounds the total size of open memory elements in any process reachable from r.

3.3 The fundamental property

The question is now to see whether it is possible to obtain a characterisation of the behaviour of a lifted process $\ell_K(p)$ solely in terms of p. Intuitively, $\ell_K(p)$ being p enriched with a mechanism for escaping computations not leading to any observable actions, one might

think that $\ell_K(p)$ is bisimilar to the transition system generated by those traces of p which lead to an observable action. This is almost true.

To give a precise statement, we need first a few notations and definitions. An RCCS transition as defined above is fully described by a tuple $t = \langle r, a, \Theta, r' \rangle$ where r is the *source* of t, r' its *target*, a its label and Θ its identifier. If $a \in K$ we say that t is a *commit* transition, otherwise it is a *reversible* transition. If $\Theta = \theta$ ($\Theta = \theta^-$) we say t is *forward* (*backward*). A *trace* is a sequence of composable transitions, and we write $r \rightarrow_\sigma^* s$ ($p \rightarrow_\sigma^* q$) whenever σ is an RCCS (CCS) trace with source r (p) and target s (q). A trace is said to be forward if it contains only forward transitions.

A final and key ingredient is the notion of causality between transitions in a given forward trace. For CCS this is usually defined using the so-called proof terms [6], but one can also use RCCS memories.

The set of memories involved in a forward transition $t = \langle r, a, \theta, r' \rangle$ is defined as $\mu(t) := \{m \in r \mid \exists a, q : \langle \theta, a, q \rangle . m \in r'\}$; this is either a singleton, if no communication happened, or a two elements set, if some did.

Definition 1 (Causality) *Let $\sigma : t_1; \ldots; t_n$ be a forward RCCS trace:*
— t_i and t_j with $i < j$, are in direct causality relation*, written $t_i <_1 t_j$ if there is $m \in \mu(t_i)$, $m' \in \mu(t_j)$ such that $m \sqsubset m'$; one says that t_i* causes *t_j, written $t_i < t_j$, if $t_i <_1^* t_j$.*
— σ is said to be causal *if for all transitions t_i with $i < n$, $t_i < t_n$; it is said to be k-*causal *if it is causal, its last transition t_n is labelled with $k \in K$, and all preceding transitions are labelled in K^c.*

One extends this terminology to CCS traces by saying a CCS trace $p \rightarrow_\sigma^* p'$ is causal, if it lifts to a *causal* trace $\ell_K(p) \rightarrow_{\sigma'}^* r'$ with $\varphi(r') = p'$.

With the notion of causal trace in place, we can define the causal compression of a process p relative to K.

Definition 2 (Relative causal compression) *Let p be a CCS process, its* causal compression relative to *K, written $\mathsf{CTS}_K(p)$, is the LTS $\langle P, p, K, \twoheadrightarrow \rangle$ where $\twoheadrightarrow_k$ is defined as $q \twoheadrightarrow_k q'$ if $q \rightarrow_\sigma^* q'$ for some k-causal trace σ.*

We are now ready to state the theorem that characterizes the behaviour of $\ell_K(p)$ in terms of the simpler process p.

Theorem 1 (Fundamental property [1]) *Let $\mathsf{TS}_K(p) := \langle R, \ell_K(p), A, \rightarrow \rangle$ be the LTS associated to the lift $\ell_K(p)$, $\mathsf{TS}_K(p) \sim \mathsf{CTS}_K(p)$.*

As said above, it is not true that $\mathsf{TS}_K(p)$ is bisimilar to the transition system of traces of p leading to observable actions, one has to be careful to restrict to causal traces. A trivial but useful rephrasing of this result is:

Corollary 1 *Let p be a CCS process, and $\mathcal{S}$ be its specification, if $\mathsf{CTS}_K(p) \sim \mathcal{S}$ then $\ell_K(p) \sim \mathcal{S}$.*

In words, this says that to check the correctness of $\ell_K(p)$ with respect to $\mathcal{S}$, it is enough to check the correctness of $\mathsf{CTS}_K(p)$.

If one goes back to the example at the beginning of this section, this says that $\ell_{\{a,b\}}((x)\big(x \mid x \mid \bar{x}.\bar{x}.a.p \mid \bar{x}.\bar{x}.b.q\big))$ is equivalent to $a.p + b.q$, as soon as the causal compression of $p = (x)\big(x \mid x \mid \bar{x}.\bar{x}.a.p \mid \bar{x}.\bar{x}.b.q\big)$ relative to $\{a, b\}$ is. This is easily seen in this example, and in fact, as often in practice, $\mathsf{CTS}_K(p)$ and $\mathcal{S}$ turn out to be equal.

The interest of this fundamental property lies in the fact that the causal compression relative to K, $\mathsf{CTS}_K(p)$, is significantly smaller than the partially reversible process $\ell_K(p)$. A natural question is therefore, given a process p, to compute $\mathsf{CTS}_K(p)$. By finding an efficient way to do this, one would obtain an efficient verification procedure. This is the object of the next section.

4 Causal compression

A first idea to extract the causal transition system of a process p is to use the LTS generated by $\ell(p)$ and screen off non causal traces. One cannot know however whether a trace can be extended into a k-causal form until a commit is effectively taken, and such an approach would likely lead to both superfluous (because lots of traces will not be causal) and redundant (because of trace equivalence) computations. A more astute approach is to look only at traces that will eventually be in a k-causal form. This requires a bottom up view of traces where one starts from commits inside a term, and then reconstructs causal traces triggering this commit by consuming its predecessors in every possible way.

However, there is no need to work directly in the syntax, and event structures [3] provide exactly what is needed here: a truly concurrent semantics that abstracts from the interleaving of concurrent transitions, and more importantly an explicit notion of causality. Among the various types of event structures the most often considered are prime ones, because consistent runs can be simply characterized. Yet they lead to quite large data structures.[1] Our algorithm uses instead *flow event structures (FES)* [6, 7, 8]. On the one hand, there is a simple inductive translation of CCS terms into FESs that incurs no computational cost; on the other hand, FES are algorithmically convenient compact forms of event structures.

We first explain how to extract the causal compression $\mathsf{CTS}_K(p)$ from the translation of p into an FES. Then we discuss computational issues such as how to make this an algorithm, and how some of the apparent computational costs can be circumvented at the level of the implementation.

4.1 Flow event structures

A (labelled) flow event structure is a tuple $\mathcal{E} = \langle E, \prec, \#, \lambda\rangle$ where
— E is a set of *events*,
— $\prec \subseteq E \times E$ is the *flow relation* which has to be irreflexive,

[1] Specifically in prime event structure causes of an event must be uniquely determined, and this forces duplication of the future of an event each time it is engaged in a synchronization.

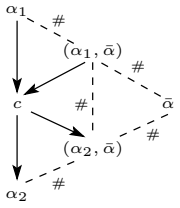

Figure 4: FES representation of $p := \alpha.c.\alpha.0 \mid \bar{\alpha}.0$. Events are named after their labels when these are not ambiguous.

— $\# \subseteq E \times E$ is the *conflict relation* which is symmetric,
— and $\lambda : E \to A$ a labelling function.
The idea is that the flow relation gives all immediate possible causes of an event, while the conflict relation indicates a conflicting choice between two events.

Definition 3 *Let $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ be an FES, a set $X \subseteq E$ is a* configuration *of $\mathcal{E}$, written $X \in \mathcal{C}(\mathcal{E})$, if it is:*
— conflict free*: $\# \cap (X \times X) = \emptyset$,*
— cycle free*: $\prec^* / X$ is a partial order,*
— *and* left-closed up to conflicts*: if $e \in X$ and there is $d \in E$ such that $d \prec e$ then either $d \in X$ or there exists $f \in X$ such that $f \prec e$ and $f \# d$.*

The last two conditions are the price to pay for working with FESs, and are not needed for prime ones. The first one will require some optimised structuring of the conflict relation, we'll return to this point soon.

A configuration X in $\mathcal{E}$ with $e \in X$ is *e-minimal* if $\forall e' \in X : e' \prec^* e$. The set of e-minimal configurations is denoted by $\mathcal{C}\langle \mathcal{E}, e \rangle$.
There is an easy inductive translation u unfolding any CCS process into a FES [6], where events correspond to communications, and configurations are those subsets of events that a trace can trigger. We will not recall here this translation, and only give an example (see Fig. 4). The correctness of u is given by the following representation theorem:

Theorem 2 ([7]) *Let p be a CCS process, and $\mathcal{T}_{\simeq}(p)$ stand for the traces of p quotiented by trace equivalence, then $(\mathcal{T}_{\simeq}(p), \leq)$ and $(\mathcal{C}(u(p)), \subseteq)$ are isomorphic.*

One can define a transition system out of an FES. To do this, we define $\mathcal{E}|X$, the *residual* of $\mathcal{E}$ by a configuration X in $\mathcal{C}(E)$.

Definition 4 (Residual) *Let $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ be an FES, X be a configuration of $\mathcal{E}$, and define $X_\# := \{e \in E \mid \exists e' \in X : e' \# e\}$. The residual of E by X is $\mathcal{E}|X := \langle E', \prec', \#' \rangle$ where:*

$$E' := E \setminus (X \cup X_\#) \quad \prec' := \prec \cap (E' \times E') \quad \#' := \# \cap (E' \times E')$$

The LTS associated to $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ has initial state $\mathcal{E}$, and transition relation given by $\mathcal{E}' \rightarrow_X \mathcal{E}''$ if $X \in \mathcal{C}(\mathcal{E}')$ and $\mathcal{E}'' = \mathcal{E}'|X$.

It is here that our reframing of the compression question in terms of event structures pays off, since to obtain the causal compression of the transition system above, all one has to do is to restrict labels to e-minimal configurations such that $\lambda(e) \in K$. The *causal LTS* associated to $\mathcal{E}$, written $\mathsf{CTS}_K(\mathcal{E})$, has initial state $\mathcal{E}$, and transition relation given by $\mathcal{E}' \twoheadrightarrow_k \mathcal{E}''$ if there is an event $e \in E'$ such that $\mathcal{E}' \rightarrow_X \mathcal{E}''$ with $X \in \mathcal{C}\langle \mathcal{E}', e \rangle$ and $\lambda(e) \in K$. As a consequence of the representation theorem one gets:

Lemma 1 *Let p be a CCS process, then $\mathsf{CTS}_K(p)$ and $\mathsf{CTS}_K(u(p))$ are isomorphic.*

At that point, we have an equivalent definition of $\mathsf{CTS}_K(p)$ in terms of the FES $u(p)$, and it remains to see how one can turn this definition into an algorithm. This is what we discuss now.

4.2 Algorithmic discussion

First, the unfolding $u(p)$ is in general an *infinite* object even if we restrict to finite state processes. To keep with finite internal data structures, we require each recursive process definition to be guarded by a commit action. This seems a reasonable constraint, in that there is a priori no reason to model a transactional mechanism with a process that allows infinite forward inconclusive traces.

To compute $\mathsf{CTS}_K(u(p))$, we use instead of u, a *partial unfolding* u^{fin} that coincides with u except it does not unfold any recursive definition. The constraint above ensures that every commit k that is reachable by a single causal transition can be seen by this partial unfolding. Only after triggering the event corresponding to k, are the recursive calls guarded by k (if any) unfolded, and their translations by u^{fin} added to the residual of the obtained event structure. One then checks whether the obtained residual event structure is isomorphic with some obtained previously, and adds it to the state space if not. Given a process p, the algorithm to compute $\mathsf{CTS}_K(u(p))$ proceeds as follows:

0. $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle := u^{fin}(p)$
1. For all $e \in E$ such that $\lambda(e) \in K$, compute the e-minimal configurations $X_e \in \mathcal{C}\langle \mathcal{E}, e \rangle$.
2. For each such X_e build the residual $\mathcal{E}|X_e$, with recursive definitions guarded by e unfolded using u^{fin}.
3. Add the transitions $\mathcal{E} \twoheadrightarrow_k \mathcal{E}|X_e$ to the CTS under construction.
4. For each residual $\mathcal{E}|X_e$ not isomorphic to any previous one, set $\mathcal{E} := \mathcal{E}|X_e$ and goto step 1.

By the representation theorem, this algorithm will terminate as soon as $\mathsf{CTS}_K(p)$ is finite.

In practice most of the isomorphism tests can be avoided by using a quite discriminative equality test between FES signatures which is linear in the number of events. Another

efficiency problem one has to deal with is the internal representation of the conflict relation (which is involved in step 1 because of the conflict-free condition on configurations). In prime event structures conflict is inherited by causality, that is to say if $e\#e'$ and $e' \prec e''$, then $e\#e''$. Hence a rather compact way to represent conflict is to keep only $(e, e') \in \#$ and deduce when needed that $e\#e''$ by heredity. We have found that a similar compact structure, which we call a conflict tree can be used for FESs. Conflict trees are built during process partial unfoldings, and result in a typically logarithmically compact representation of conflict, for a low computational cost. An example of a conflict tree is given Fig. 5: conflicts is predicated of intervals, and $[n-m]\#[n'-m']$ means that any pair of events indexed within $\{n, \ldots, m\} \times \{n', \ldots, m'\}$ is in conflict.

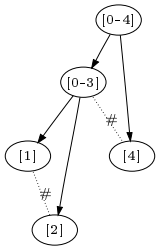

Figure 5: Conflict tree of $a_3.(b_0 \mid c_2 + d_1) + e_4$

5 Causal module and tests

The relative compression algorithm was implemented as an Ocaml [9] library Causal [10]. Having a library instead of an independent tool allows to use the underlying language that offers more construction primitives than CCS. Any interesting encoding needs parametric process definitions in order to define systems with varying number of agents, and our module offers simple CCS process constructors, so that one has a real programming language to build large processes.

5.1 Benchmark

To get a sense of how well our verification technique performs compared with a straight bisimulation based verification, we ran several tests[2] using encodings of the dining philosophers problem. This timeless example of distributed consensus involves n philosophers eating together around a table. Each of them needs two chopsticks to start eating, and has to share them with his neighbours. When a philosopher has eaten, he releases his chopsticks after a while and goes back to the initial state. In the partial implementation, say p_{part}, once a

[2] Tests were made with an Intel Pentium 4 CPU 3.20GHz with 1GB of RAM.

philosopher takes a chopstick he never puts it back unless he has successfully eaten. In the fully correct one, say p_{full}, he may release chopsticks at any time (thus avoiding deadlocks). The CCS processes p_{part} and p_{full} for $n = 2$ correspond roughly to the earlier examples (1) and (2). (See [1] for a general definition and detailed study.)

There are two main reasons for taking the dining philosophers example. First it is a paradigmatic example of distributed consensus, so the way to solve it without access to the scheduler (by adding additional semaphores for instance) has to involve backtracking. Second, it turns out that the number of possible states of the specification is given by a Fibonacci sequence[3]

$$S(1) = 1 \quad S(2) = 3 \quad S(n+1) = S(n) + S(n-1)$$

This is convenient in that it gives a simple means to compare the time of computation with the size of the specification state space. Verifying correctness of p_{full} using the Mobility Workbench (MWB) [11] (see Fig. 6) proved to be impossible beyond 5 philosophers (around

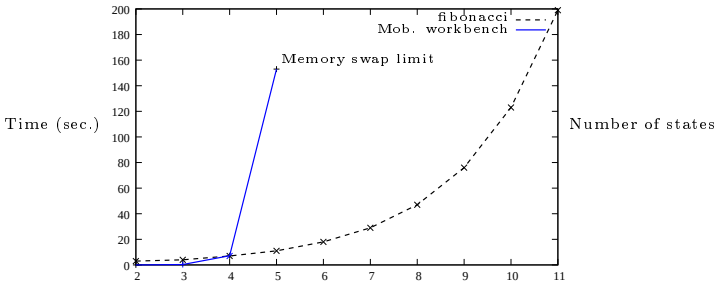

Figure 6: Direct bisimulation test for p_{full}.

160 specification states) because of memory limitations. By using first the Causal module (see Fig. 7) to extract the causal transition system of p_{part}, we could verify up to 19 philosophers (around 15,000 specification states) within a time which stayed roughly proportional to the number of states. Since CTS(p_{part}) is in this case equal to the specification, the remaining part of the correctness proof takes negligible time (MWB needs 0.4s for 10 philosophers).

6 Conclusion

We have proposed a method for the verification of distributed systems which uses an algorithm of relative causal compression. The method does not always apply: the process one wants to verify must use a generic backtracking mechanism. This may seem a limitation, but it often obtains a much simpler code, and many examples of distributed transactions lend

[3] Thanks to Hubert Krivine (LPTMS) for showing us this nice result.

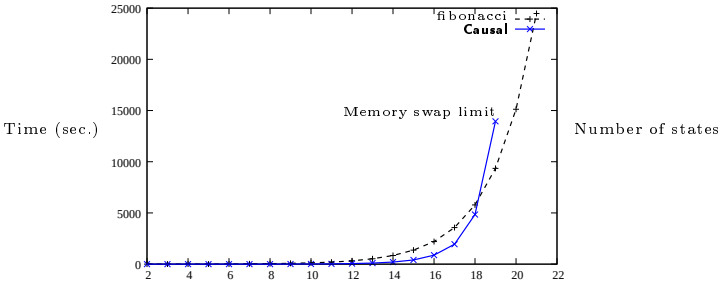

Figure 7: Relative causal compression using the Causal module.

themselves naturally to this constraint. When the method does apply, however, it proves very effective as we have shown in the dining philosophers example.

State space explosion in automated bisimulation proofs is a well known phenomenon, and trace compression techniques have been proposed to avoid the redundancy created by the interleaving of transitions [6, 12], and used in model-checking applications [13, 14]. These compressions preserve bisimilarity, whereas our does not, and is of a completely different nature. Besides, and because our algorithm uses event structures, we also cash in on this classical kind of compression.

There is no reason why this verification method should be limited to CCS. Other concurrent models can be equipped with backtracking, and forward and backward aspects of correctness can be split there as well. Recent work extends the concept of partially reversible computations to various process algebras [15], and it is possible to define an analogue of RCCS for the π-calculus. New advances in event structure semantics for π-calculus [16] might allow to extend the causal compression algorithm, so as to cover the important case of name-passing calculi.

References

[1] Vincent Danos and Jean Krivine. Transactions in RCCS. In *Proceedings of CONCUR'05: 16th International Conference on Concurrency Theory*, volume 3653 of *LNCS*, 2005.

[2] Vincent Danos, Jean Krivine, and Fabien Tarissan. Self assembling trees. In *Proceedings of the 7th International conference on Artificial Evolution (EA'05)*, 2005. To appear.

[3] Glynn Winskel. Event structure semantics for CCS and related languages. In *Proceedings of 9th ICALP*, volume 140, pages 561–576, 1982.

[4] Robin Milner. *Communication and Concurrency*. International Series on Computer Science. Prentice Hall, 1989.

[5] Vincent Danos and Jean Krivine. Reversible communicating systems. In *Proceedings of CONCUR'04: 15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 292–307, 2004.

[6] Gérard Boudol and Ilaria Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 411–427, 1989.

[7] Gérard Boudol. Flow event structures and flow nets. In *Proceedings of LITP Spring school on Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 62–95, 1990.

[8] Rob van Glabeek and Ursula Goltz. Well-behaved flow event structures for parallel composition and action refinement. *Theoretical Computer Science*, 311(1-3):463–478, 2003.

[9] The Ocaml programming language. Available at http://caml.inria.fr.

[10] Causal — ocaml module for causality analysis of CCS processes. Available at http://pauillac.inria.fr/~krivine.

[11] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π-calculus. In *Proceedings of CAV'94: Computer-Aided Verification*, volume 818 of *LNCS*, pages 428–440, 1994.

[12] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of CAV'91: Computer-aided verification*, volume 575 of *LNCS*, pages 332–342, 1991.

[13] Alessandro Bianchi, Stefano Coluccini, Pierpaolo Degano, and Corrado Priami. An efficient verifier of truly concurrent properties. In *Proceedings of Parallel Computing Technologies*, volume 964 of *LNCS*, pages 36–50, 1995.

[14] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV'04: Computer-aided verification*, volume 3114 of *LNCS*, pages 484–487, 2004.

[15] Iain Phillips and Irek Ulidowski. Reversing algebraic process calculi. In *Proceedings of FOSSAC'06*, LNCS, 2006. To appear.

[16] Daniele Varacca and Nobuko Yoshida. Typed event structures and the π-calculus. In *Proceedings of MFPS XXII*, 2006. To appear.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399


INRIA